In the Specification

Please amend the specification as follows:

On page 3, lines 7-14 have been amended as follows:

--The event control block is a real structure of an event and is formed of a data structure managed by the kernel. Creating an event control block means that a task creates an event of its own. In the case that the second task 2 with the priority higher than that of the first task is created and started to execute and the second task 2 calls the kernel system function of waiting for (receiving) an event, then since the first task transfers no event yet, the second task 2 is blocked to the wait state waiting for the event to be transferred (sent) and is queued into the waiting-list of the event control block 1.--

On page 3, line 20 – page 4, line 8, have been amended as follows:

--Thereafter, when the first task starts to send the event, at first the second task receives the event, is woken up and is resumed execution. Namely, the event is first transferred to the second task 2 first queued to the waiting-list based on the FIFO (First-In-First-Out), so that the second task is executed. Therefore, it causes the third and fourth tasks 3 and 4 having relatively higher priorities not to be first performed. Namely, the above-described method may be adapted to a known round robin scheduling method, but it may not satisfy the scheduling mechanism which supports real time characteristics.

In order to overcome the above-described problem, the structure of the waiting-list should be formed in a doubly linked list, and when an event is sent, the task with the highest priority should be searched in the waiting-list and resumed to execute. However, in this case, since it is not until an event is sent that the searching of the waiting-list is performed, it may take more reaction-time. And the more number of the tasks waiting for the event, the more time it takes also.--

On page 6, lines 8-13 have been amended as follows:

--As shown in Figure 2, in this example, it is defined that the priority value of the first task is 40, the priority value of a second task 102 is 30, the priority values of third and fourth tasks 103 and 104 are 20, respectively (the lower priority values have the higher priority), and that the second through fourth tasks 102 through 104 are trying to receive the event which the first task transfers periodically through an event control block 101 of the first task.-

On page 6, line 22 - page 7, line 19 have been amended as follows:

--The task order queue in the waiting-list at the time when the second through fourth tasks 102 through 104 have been all blocked (E in Figure 4), is not the sequence of calling the kernel system function as shown in Figure 1. Instead, the order will be the third task 103, the fourth task 104 and the

second task 102 which is the priority order of tasks as shown in figure 2. When a kernel system function call is performed for receiving an event by each task but no event is sent yet, each task is blocked and is set to the wait state. At this point, the priority of each task is checked and the task is inserted into the position of the priority order of the waiting-list so that the waiting-list becomes the priority order. Therefore, the waiting-list is always maintained in the state of the priority order such that the task with the highest priority is placed at the head of the waiting-list.

When the first task calls a kernel system function of transferring (sending) the event from the point F of Figure 4, the priority check is not additionally needed with respect to the tasks of the waiting-list. The task with the highest priority at the head of the waiting-list is picked up from the list, and the event value is transferred to the task, so that the execution of the task is resumed. It means that since the waiting-list is already aligned in the higher priority order, when transferring the event, the task with the highest priority of the waiting-list first obtains (receives) the event by merely waking up and resuming the head-positioned task.

As shown in Figure 4, the event transferred by the first task is alternatively obtained by the third task 103 and the fourth task 104 having the highest priority equally, and the second task 102 which has the relatively lower priority does not obtain the event.--

On page 8, lines 11-20 have been amended as follows:

--As a result of the check, if the event ID is invalid, it means an error situation exists where the event-receiving attempt is performed with respect to the non-existing event. Therefore, the scheduling is enabled in Step S12, and the current task is returned from the kernel system function with an error code in Step S13.

As a result of the check, if the event ID is valid, next it is checked that whether the event value has been already transferred (sent) in Step S4. If the value exists, the event value is obtained from the buffer in the event control block in Step S5, the process routine is performed by the kind of the event in Step S11, the scheduling is enabled in Step S12, and the current task is returned from the kernel system function with the event value in Step S13.--

On page 9, lines 9-13 have been amended as follows:

--/* if the waiting-list is empty (any task that waits for a corresponding event does not exist), or if the priority of the current task is lower than the priority of the tail portion task of the waiting-list or is the same (the priority of the current task is lower than any other tasks already queued with the standby list), the current task is directly inserted into the rear end (tail) of the waiting-list. */--

On page 11, line 21 - page 12, line 1 have been amended as follows:

--The head (leading) task of the waiting-list, namely, the task having the highest priority among the tasks of the wait state is woken up and is resumed in its execution when another task which is supported to send the event calls a kernel system function of sending the event or the time-out is elapsed to thereby cause the preemption. The task routine is returned from the kernel system function of receiving the event based on the steps S11 through S13.

On page 12, lines 4-19 have amended as follows:

--When a certain task which is currently being executed calls the kernel system function of sending the event in Step ST1, the scheduling is temporarily disabled like the kernel system function of obtaining the event in Step ST2, and it is checked whether the argument ID of the event that the current task sends to is valid in Step ST3. As a result of the check, if the event ID is invalid, it means that an error situation exists where the event-sending attempt is performed with respect to the non-existing event. Therefore, the scheduling is re-enabled in Step ST12, and the current task routine is returned from the kernel system function with the error code in Step ST13.

As a result of the check, if the event ID is valid, next it is checked whether the waiting task exists in the waiting-list of the event in Step ST4. if any task of the wait state does not exist with respect to the event, the event value is simply stored in the event buffer of the event control block in Step ST5, and the event kind-based process routine may be additionally performed in

Step ST11. Then the scheduling is re-enabled in Step ST12, and the current task routine is returned from the kernel system function in Step ST13.--

On page 13, lines 12-15 have been amended as follows:

--Thereafter, if the scheduling is re-enabled in Step ST12, the preemption can be occurred, so that the task (which was the head (leading) task in the waiting-list), which has been set ready to wake up in Step ST10 is resumed in its execution.--